Yevheniya Nosyk

MIGRATION OF A LEGACY WEB APPLICATION TO THE CLOUD

Bachelor's thesis
Degree programme in Information Technology

2018

South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree	Time
Yevheniya Nosyk	Bachelor of Engineering	February 2018

Title

Migration of a legacy web application to the cloud

51 pages
0 pages of appendices

Commissioned by

NeosIT Services GmbH

Supervisor

Matti Juutilainen (XAMK), Warren van der Woude (NeosIT Services GmbH)

Abstract

The idea of this thesis work was to study the area of containerization and cloud computing. Nowadays these technologies are gaining popularity very rapidly and becoming de-facto standards in information technology. The skills gained during the implementation of this thesis are necessary to be up to date with the current trends in IT industry.

This thesis was commissioned by the company called NeosIT Services GmbH. It is a Germany-based enterprise specialising in the creation of high transaction platforms. The company has experience in cloud computing and that is where the idea for the thesis topic came from.

Before doing the practical part, a theoretical study was conducted. It was important not only to understand how microservices and the cloud are functioning nowadays, but also to look back in the history and understand why they appeared. When it comes to cloud computing, the most typical task is to containerize an application and move it to the cloud. For this thesis, a simple Silex application was created in a monolithic (legacy) way. It was then converted into API. The refactored application was containerized and moved to a cloud service provider – Amazon Web Services.

This thesis covered the main aspects of cloud computing and microservices as well as how they were implemented in practice. Ideally, applications should be developed with the cloud in mind. However, this thesis described the workflow of a typical working-life case – an old-style application already existing without time and resources to build a new one. The ability to refactor legacy applications is a good starting point when moving businesses to the cloud.

Keywords

Containers, cloud, Docker, AWS, PHP, Silex

CONTENTS

1 INTRODUCTION

Information technology has been developing rapidly in the first decade of the 21st century. The capacity of computers is growing exponentially as predicted by Moore's law and the rate of changes is soaring. There is a need to control this blast of capacity. That is why the service orientation and the cloud are now the two leading technologies in IT. For decades, companies were investing in their own hardware and delivered applications as services running on their premises. This solution was expensive and inefficient.

Recently, a new approach appeared – cloud computing. The cloud is designed as a pool of resources where customers pay only for the capacities they use. (Waschke 2015.) This means that software developers do not have to think about the underlying architecture and can now concentrate on building cloud native applications. To make the deployment of these applications simpler, microservices and containers emerged. The idea is to break down a monolithic application into smaller parts and move each of them to a separate container. Each container is a runtime environment with just enough of the operating system resources to run the given piece of code. This allows scaling applications seamlessly, moving them between different virtual machines and, finally, deploying them easily to the cloud.

It is difficult to underestimate the influence that cloud computing and microservices have over the IT community. Figure 1 describes the growth of cloud computing by comparing the amount of money spent on it from 2015 and predicting the number in 2020.

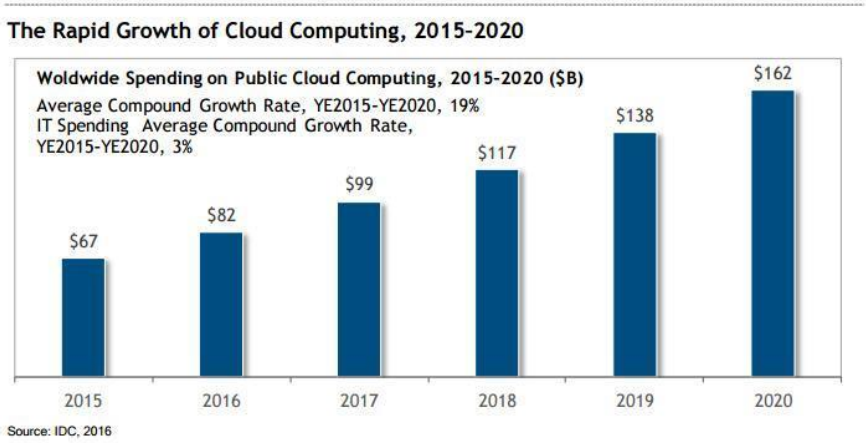

Figure 1. The rapid growth of Cloud Computing, 2015 – 2020 (Columbus 2017)

The cloud is the fastest growing IT industry now (CloudAcademy 2018). Many companies are moving their businesses to the cloud and they are constantly looking for cloud administrators and software developers that can create service-oriented applications.

The topic of the thesis is to migrate a legacy web application to the cloud. According to Orban (2016), there are six migration approaches:

- Rehost – change the underlying hardware of the application and its infrastructure.
- Replatform – upgrade/modify some of the legacy application's components, before it can be moved to the cloud.
- Repurchase – move to another software as a service (SaaS) platform.
- Retire – get rid of the application, if it no longer serves its purpose or refactoring will be expensive
- Retain – do not migrate if the current architecture does not cause problems, such as poor performance or cost-inefficiency.
- Refactor – modify the application's structure from the monolithic one to service-oriented.

The last approach was chosen for this thesis. The idea was to create a monolithic application using the Silex PHP microframework. This is the way that most legacy web applications are built. Then, the application was converted into an API. Its components (the application itself and nginx) were moved to separate containers and linked. Amazon Web Services was chosen as a cloud platform where the whole project was running.

This work was commissioned by a Germany-based company called NeosIT Services GmbH. This enterprise has been developing platform-independent solutions for a while and has a solid background in cloud computing and containerization. Together with the company, a real-life case was developed. An experienced company supervisor not only made sure this thesis work was done correctly from the technical point of view, but also gave advices on how a similar case is expected to be managed in the working life.

This report consists of seven chapters. The introduction gives an overview of what this thesis is about. The Cloud and Microservices chapters provide a theoretical background information needed to deeply understand the topic. Chapter 4 introduces technologies and tools used in the implementation part.

Chapter 5 describes the practical part. Finally, the future trends in cloud computing are introduced in Chapter 6 and the whole work is summed up in Chapter 7.

2 MICROSERVICES

The concept of microservices became extremely popular a couple of years ago and can be defined as a certain way of software design that breaks down a monolithic application into multiple service components. Each process is then run separately. It greatly speeds up the development and deployment, as teams of programmers can concentrate on working on a certain feature without the need to rebuild the whole application every time changes are made. (Lewish & Fowler 2014.) However, not every single application should be built with microservices in mind. The purpose of this chapter it to look at the history of microservices, analyse their advantages and disadvantages and to see which technologies are used to implement them in practice.

2.1 History

Even though the term “microservices” has appeared quite recently, the idea of the separation of concerns in not new. It took microservices decades to become a standard of the software development. Looking back in the history will help to understand how microservices have evolved and which problems they have solved.

According to Tozzi (2017), the history of microservices goes back to the 1960s when object-oriented programming started to evolve. Even though the monolithic applications were dominating, the developers were looking for the ways to break down a single piece of software into smaller components. Later in the 1980s, a new concept of operating system design appeared – microkernels. The traditional monolithic kernel was split into multiple smaller parts to help manage the increasing complexity of the operating systems. Even though, because of the performance issues they never became popular outside scientific circles, computer scientists were thinking about service-based architecture already at that time. (Tozzi 2017.)

The modern microservices architecture was greatly influenced by SOA – Service Oriented Architecture. It became popular in the early 2000s and that was the first time the separation of concerns was introduced. The idea behind SOA is to have a set of services that have their own interfaces and are delivered over the network using protocols. Microservices can be seen as successors of service-oriented architecture. The term itself appeared first in 2011 to describe a set of architecture patterns and ideas. Nowadays they have become a very hot trend aiming at creating scalable and highly maintainable software. (Dragoni et al. 2017, 6.) The microservices are best defined by Fowler (2014) as follows: “The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.”

2.2 The history of containers

Together with the development of SOA, there was a growing need to increase the performance and response time. Running microservices on regular machines could not fully benefit from their advantages. Instead, it would be better to have an isolated execution environment for each service with just enough of operating systems’ resources. That is why there was a need for what is now called “containers”.

The first technology aiming at the isolation of applications on the same machine was called chroot. It was developed in 1979 and hardly resembled modern containers, but already made it possible to control which files of the operating system could or could not be accessed. This concept was greatly improved in 2000, when the FreeBSD operating system was developed. It included a feature called jails which isolated processes based on administrator’s needs. (Tozzi 2017.) Linux VServer was another enhancement of the jails architecture that isolated computer resources. It was introduced in 2001.

The first time the term “container” appeared was in 2004, when Solaris Containers were introduced. The container feature became a part of the Solaris operating system. System administrators could configure “zones” to define which resources a certain application could use. OpenVZ framework for Linux, introduced in 2005, had the same purpose as Solaris Containers and served to isolate users, files, processes etc. (Tozzi 2017.)

Linux Containers (LXC) were developed in 2008 as a container framework for the Linux operating system. They functioned in a way to provide dedicated Linux environments on a single host machine. They also benefited from using a host operating system and its kernel. Without a guest operating system, they start much faster than virtual machines. LXC technology was used as a basis for different container management systems. Some of the most widely used nowadays are Docker and Rocket. They will be discussed in more details in the following chapters. (Tozzi 2017.)

2.2.1 Container advantages

Advantages of containers are often discussed in the context of virtualization and the difference between the two. Figure 2 shows simplified models of the underlying infrastructure of virtual machines and containers. Even though some of the characteristics are rather similar, there are certain differences. For example, in virtualization there is always the overhead of a guest operating system. Together with a hypervisor, a virtual machine can take up several gigabytes and become resource intensive. However, it also means that virtual machines are independent of a host operating system and use their own kernel. While virtual machines virtualize the underlying hardware, containers virtualize the underlying operating system. They depend on their host operating system and containers on the same machine share the same operating system. Containers are also considered more lightweight. (Medium 2016.)

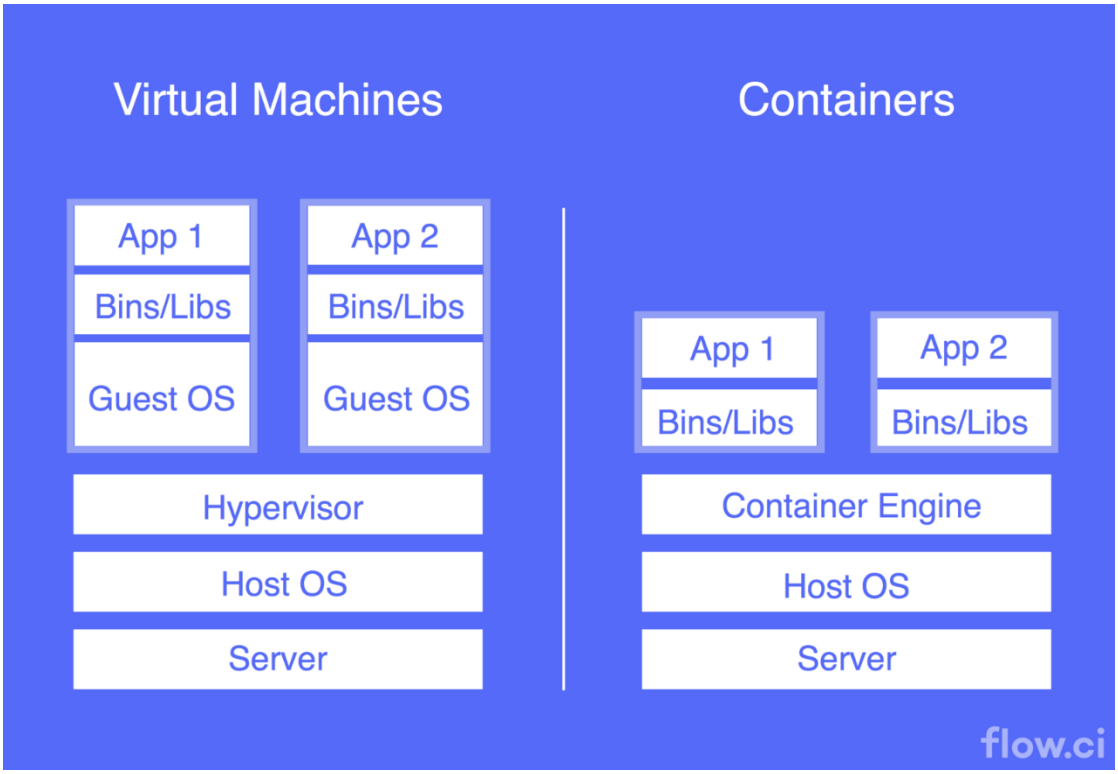

Figure 2. Simplified model of virtual machines and containers (Medium 2016)

Other reasons why more and more IT engineers are using containers in software development are the following:

- Consistency– container is an isolated environment that includes everything needed for an application to run, including, for example, dependencies and libraries. This makes sure that applications will be consistent, regardless of the platform where they will be deployed.
- Environment independence – containers can be deployed on different platforms, including Linux, Windows, Mac and even virtual machines.
- Isolation – each container has its own piece of host operating system's recourses, such as CPU, memory, storage and network. Multiple containers on the same host are also logically isolated from each other and from the operating system itself. (Google Cloud Platform 2018.)
- Speed – creating, replicating and deleting containers takes place in less than a second. An operating system does not need to boot, so starting a container is very quick. It speeds up greatly the workflow of web developers. (Kumina 2017.)

Containerisation brings a lot of benefits to software developers themselves. It is true that when they assume that applications can run seamlessly in different environments and multiple cross-platform tests are not needed anymore, they will be able to spend more time on developing new functionalities.

2.2.2 Container disadvantages

When containerization became popular, some software developers were sure that it was time to move all the old applications to containers. However, it is important to remember that any single technology cannot be suitable for all the situations. Containers' disadvantages, listed below, were defined by Linthicum (2018):

- Security is still under question – containers do not have the same level of isolation than virtual machines. If a hacker finds a way to break into a host operating system, containers on the same machine will also be compromised. In the same way, if a vulnerability is found in a container, an underlying operating system can also be hacked. While it is important to secure containers from the technical point of view, it is also necessary to teach the staff the basics of information security. Containerization is a relatively new technology and people may have no experience working with it.
- Not suitable for all the application – containers make applications more complex, as there are many parts that must be bundled together. Usually, if an application does not need to be scaled, it can be built in a monolithic way. Containers are suitable for situations when a part of an application should be modified. This part (which is actually a separate container) will then be redeployed without touching the rest of the code.
- Do not always replace a virtual machine – as it was described in the previous chapter, virtual machines are more isolated from an underlying operating system. Each of them has its own operating system and other components. They can be migrated easily to virtually any system. On the other hand, containers are more coupled to the operating system because of multiple dependencies. That is why they are not as portable as virtual machines.
- Can become messy – containers are launched very quickly, so there is a risk to deploy too many of them and loose the count. Unused containers should be shut down, otherwise, they will continue consuming resources. If they are hosted on a cloud provider, this waste of resources can become expensive. (Linthicum 2018.)

In general, containers do provide what is promised – fast deployment, good performance, development frameworks, large community etc. However, it is too early to forget virtual machines, as containers cannot replace them yet. Also, as it was mentioned earlier, containerization is not the best solution for every virtualization task. (Bigelow 2018.)

2.3 Container management platforms

To ease the process of creating, deploying and managing software containers, there are container platforms. Even though Docker is probably the first thing that comes to mind when talking about containerization, it is not the only tool available to manage containers. Other popular container platforms will be described in this chapter.

2.3.1 LXC (Linux Containers)

LXC (short form of Linux Containers) was introduced in 2008. It is defined by Khan (2017) as “an operating-system-level virtualization to run multiple isolated Linux systems (called containers) on a host using a single Linux kernel”. It appeared for the same reason that all the other container platforms: a need for an alternative to recourse-consuming computer virtualization that would create isolated runtime environments. Firstly, it was achieved by namespaces and cgroups. Since kernel 2.6.24, a tool called Linux Containers was integrated in the Linux Kernel. It is now one of the oldest containerization technologies that shaped the way modern containers look like. (Ivanov 2017.) As it was mentioned in previous chapters, LXC is also considered a direct predecessor of Docker.

Initially, LXC was created as an alternative to virtualization, because it could host virtual machines without the overhead of a hypervisor. The advantage is that it does not necessarily have to virtualize a whole operating system inside. It was the first time a single application could run in a virtual environment. Linux Containers literally started container revolution and that is where CoreOS and Docker came from. (Jain 2016.)

2.3.2 Rocket

CoreOS was designed as a lightweight operating system for containerization. At the beginning, both Docker and CoreOS were developing in the same direction. However, Docker was becoming more and more complex, including many new features like clustering, networking etc. CoreOS, on the other hand, wanted to preserve simplicity and started working on its own alternative to Docker. The objective was to create a tool that would be fast, secure, modular

and able to distribute images. That is how Rocket (also known as rkt) was created. (Smiler et al. 2016.)

Rocket was developed as a standalone tool, having fewer dependencies and constraints. That makes it different from Docker. Usually, in case of crash, all the Docker processes would die, while in Rocker only some of them will be affected, because all the processes are separated and assigned their own personal identification numbers (PID). Rocket is suitable for running inside its native CoreOS. The commands are very similar to those of Docker, but due to the continuous development they may change. (Mocevicius 2015.)

2.3.3 Docker

As it was mentioned earlier, Docker is now a de facto standard in containerization and the most popular container platform. Turnbull (2014) defines it as “an open-source engine that automates the deployment of applications into containers.” Docker was created by a French company called dotCloud. It provided PaaS hosting for web applications. They relied heavily on LXC but already started developing their proprietary containerization solution. Until 2013 Docker was used only in the company’s internal operations but then it was released as an independent product and the company itself was renamed to Docker Inc. At the same time, Docker Code was also developing. Its aim was to ease integration with Linux Containers. Then, it moved to a different framework and started developing independently. Some very important features (among those container networking, data volumes and orchestration) were introduced. These changes were necessary to use Docker at an enterprise level. As for the company, its revenues soared dramatically when Docker was introduced. (Tozzi 2017.)

The popularity of Docker can be seen on Figure 3. It shows what has changed during a 3-year period (2014 - 2017). Evidently, the numbers of Docker hosts, applications, contributors and image pulls rose significantly. Docker Inc. also attracted some highly-qualified employees.

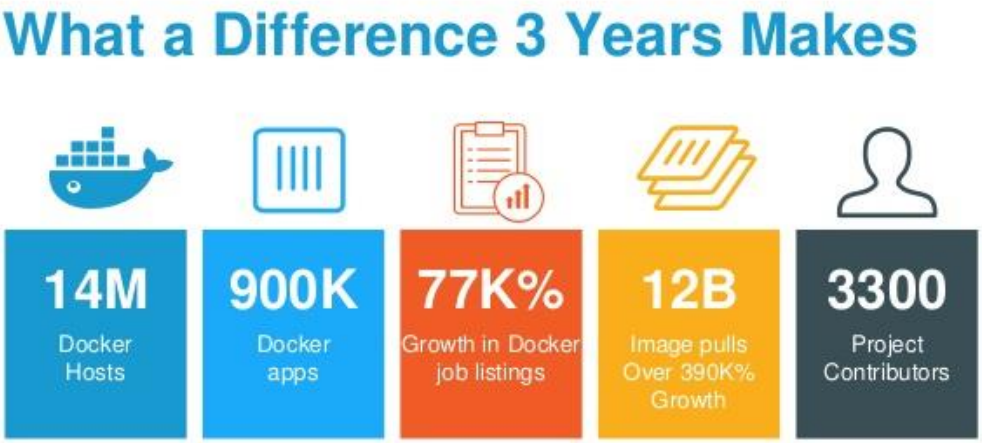

Figure 3. Development of Docker between 2014 and 2017 (Golub 2017)

Docker is valued for its ability to create, destroy and deploy containers rapidly. Containers share the resources of a host operating system, which makes them very lightweight. Docker containers are portable and can be run virtually in any environment. They are also supported by all the major cloud computing providers, e.g. Amazon Web Services (AWS) and Google Cloud platform. Docker is mostly used for deploying Java microservices and testing (especially in continuous integration and continuous deployment). Deployment and scaling applications has never been so easy. (Krochmalski 2017.)

Even though everybody is so obsessed with Docker, it does have certain downsides. For example, it is a common practice to fetch random images from Docker Hub Registry without checking that they are safe to use. There are no tools to determine where an image is coming from and what is installed there. (Negus 2015.) Docker containers are faster than virtual machines, but still they don’t have a speed of bare-metal servers. Data storage is challenging because containers must be saved before being shut down. To sum up, Docker in not the only solution for application deployment. Sometimes it makes sense to use virtual machines or bare-metal servers. Docker works best with applications that are run as a set of microservices. (Tozzi 2017.)

3 THE CLOUD

Cloud computing is another hot trend in information technology. It is used for different purposes, including streaming media, hosting services, backups, testing etc. More and more software developers build “cloud-native” applications and companies prefer to use the cloud rather than invest in hardware. Cloud is best defined by the National Institute of Standards and Technologies as “a model for rapidly deployed, on-demand computing resources such as processors, storage, applications, and services” (Washke 2015). There are various cloud deployment and delivery models as well as providers. The following chapter discusses them in detail.

3.1 The history of cloud computing

It is often said that cloud computing is a new way of doing business. However, looking back at the history shows that this concept has already been in use for a while. (Yang & Liu 2013.) The cloud computing evolution is illustrated in Figure 4. In the 1950s the data centres started to evolve, and it was the first time the computer services could be rented. The hardware was rather expensive at that time. However, the technology developed to become what is now known, a cloud computing. The technologies that played the biggest role in the cloud development are distributed systems, virtualization, Web 2.0 and service orientation. (Thamarai Selvi et al. 2013.)

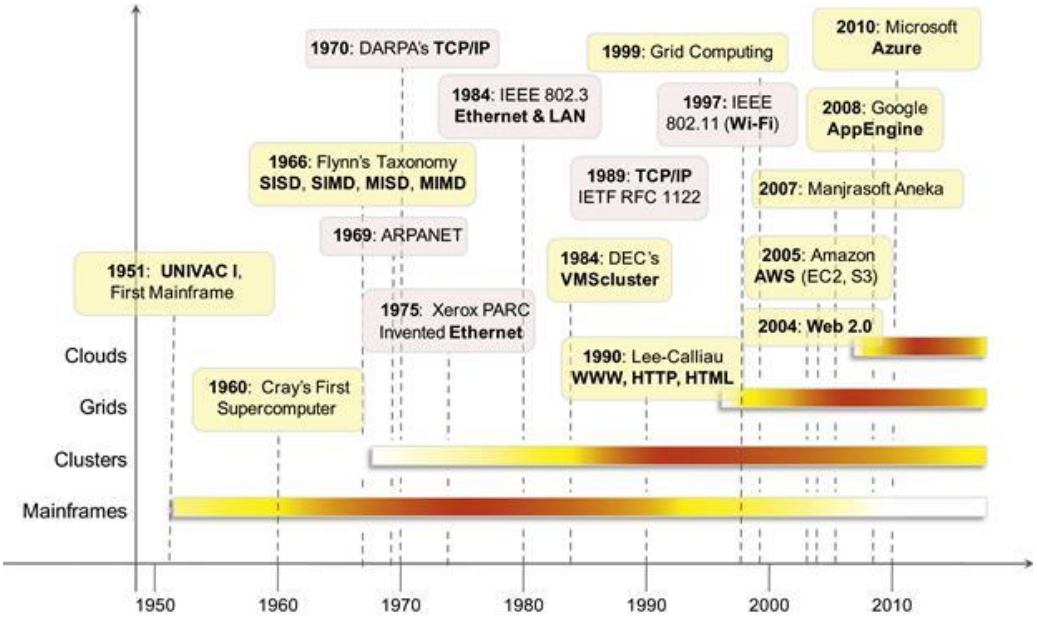

Figure 4. The evolution of distributed computing technologies, the 1950s–2010s (Thamarai Selvi et al. 2013)

At the beginning, there were standalone computers each serving their purpose and doing one single task. The computers then developed to be running multiple services on the same machine simultaneously. Still, they were all working independently from each other. It is later that the networking technology evolved and connected multiple machines in a way that they looked like a single system. (Burns 2017.) At the same time, since computers were not able to share resources between multiple users, IBM invested in time-sharing solutions to overcome this limitation. Nowadays virtualization is an effective tool to abstract hardware resources and to use them in a more efficient way.

Back at the beginning of the 2000s, Web 2.0 came into play. The Web is the way how cloud services are delivered through the Internet and can be defined as a set of technologies and tools that enable information sharing, collaboration and application composition. Due to the use of XML, AJAX and Web Services, the Web brings interactivity and dynamics to web pages, continually improving them for the end users. Web pages are now well accustomed to using cloud-based applications, such as Google Maps, Facebook, Twitter, Wikipedia etc. which is how Web 2.0 helps in promoting cloud computing as a paradigm. (Burns 2017.)

Service-oriented architecture (SOA), discussed earlier in the context of microservices, is also the core reference model for cloud computing systems (Burns 2017). In SOA, everything is delivered as a service. This technology was developed in 2005 by Chang, He, and Castro-Leon and quickly evolved to what is known now as cloud computing. It became clear that developing applications as services would simplify their management and infrastructure, abstracting the software and underlying architecture. The technologies that contributed to the development of SOA were XML and WSDL. (Yang & Liu 2013.)

3.2 Cloud computing today

Nowadays cloud computing is a focus of information technology. It is a hot trend that is developing continuously. As illustrated on Figure 5, some of the biggest players of the IT market have turned their operations towards cloud

computing. In case of Microsoft, they created cloud versions of their existing products that had already had very wide coverage. Google benefits from many individual users. They have launched Gmail, Google Docs and Google apps. All these cloud services are now widely used. Amazon has built an advanced infrastructure while being a popular shipping platform and now they also sell scalable storage, database, web hosting and e-commerce services. VMware has always been a part of cloud computing industry and a pioneer in computer virtualization. (Li et al. 2013.) Big companies always dictate the direction in which a certain industry will develop. That is why it is clear that learning cloud computing is necessary to keep up with the development of information technology.

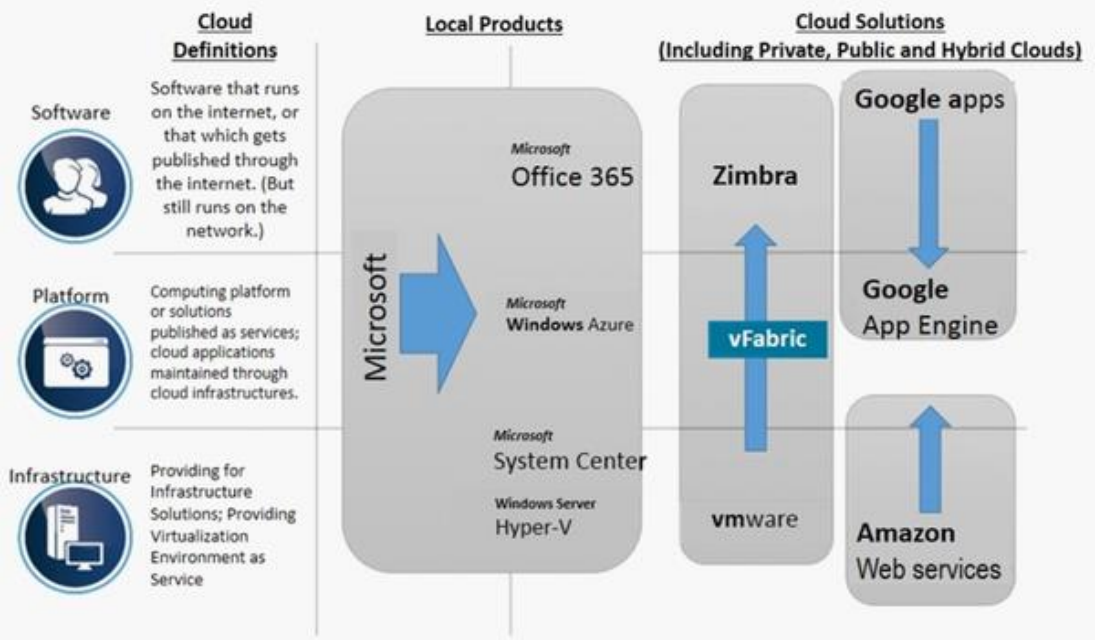

Figure 5. IT Industry leaders' development paths into the cloud (Li et al. 2013)

3.3 Cloud deployment models

Clouds are the building blocks of cloud computing. They provide an infrastructure on top of which services are delivered to customers. There are many ways to classify clouds. One of them is to define their boundaries – deployment models. (Thamarai Selvi et al. 2013.) Four of them are described in this chapter.

3.3.1 Public

Public clouds were first to appear because they literally represent the idea of cloud computing – providing resources to everyone from anywhere. The idea was to create a solution that would minimize infrastructure expenditures while offering stable performance. It eliminated the need of investing in on-premises equipment and allowed to pay only for the recourses consumed. It is the responsibility of a cloud provider to build and maintain underlying infrastructure (Mahmood 2013). In practice it means that public cloud provider owns several connected data centres and serves multiple customers. This is also referred to as multitenancy. (Thamarai Selvi et al. 2013.)

One of the reasons why public clouds became so popular is because of their competitive price. It is achieved by economies of scale. Big cloud providers have enough funds to invest in hardware, infrastructure and bandwidth. They have many customers that use cloud resources in different ways and at different times, which is called over-subscription. (Goetsch 2014.) Another source of income is, of course, advertising.

Public cloud is general-purpose and can be used for all kinds of services. However, it is available to everyone, so security can be compromised. Another concern is that there is no control over the underlying hardware. Therefore, public cloud is not suitable for sensitive data. (Goetsch 2014.)

3.3.2 Private

Large companies, especially those having a constantly high demand of IT resources, often need their own private clouds. Cloud data centers can be located either within a company (where cloud infrastructure can be built on top of the existing hardware) or provided by an external private cloud provider (e.g. Amazon Private Cloud, Red Hat Cloud Infrastructure etc.). Cloud management is either outsourced or handled by IT department of the company. (van den Berg 2015.) With the private cloud, it is possible that the same enterprise acts as a cloud provider and a cloud consumer (Mahmood 2013).

One of the problems associated with private cloud is initial investment cost. To build company's own private cloud, most probably the existing infrastructure has to be reorganized and some of the hardware needs to be changed. Sometimes it will be cheaper to rent a data center from an external private cloud provider. Furthermore, managing capacity utilization becomes even more critical as all the unutilized resources will become an additional cost to a customer. Finally, scaling a physical infrastructure is more complicated than scaling a virtual machine. (Dufficy 2014.)

3.3.3 Hybrid

Public clouds have a well-developed infrastructure and enough capacity to serve multiple customers simultaneously. They are chosen by some companies that do not have financial resources to build and manage their own data centers (private clouds). Security limitations and lack of control over the underlying hardware are two biggest concerns about public clouds. These problems can be overcome with private clouds that keep data inside a company and use an on-premises infrastructure. However, private clouds are rather expensive to build and difficult to scale. In order to combine the advantages of public and private models, a hybrid cloud was created. It allows to keep sensitive data on-premises and scale by using external resources. (Thamarai Selvi et al. 2013.)

An ideal use case for a hybrid cloud is an e-commerce platform. Traffic is changing all the time, so it is better to use an external infrastructure and pay only for the resources consumed. At the same time, sensitive data about customers and their private information will be kept on-premises for security reasons. (Goetsch 2014.)

Hybrid cloud is currently heavily used by VMware and Microsoft. They were first to understand that businesses will not move their operation to the cloud completely. Hybrid deployment model is a good compromise between the cost of public cloud and the security of private cloud. (van den Berg 2015.)

3.3.4 Community

Community cloud is characterized by The National Institute of Standards and Technologies as: “The infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or a third party and may exist on premise or off premise” (Thamarai Selvi et al. 2013). This deployment model also combines public and private clouds. Community clouds are used in public sector, scientific research, healthcare and media industries. They facilitate cooperation between community members and benefit from openness, convenience, sustainability and fault tolerance. (Thamarai Selvi et al. 2013.) This is the newest deployment model that aims to deliver efficient services while reducing associated costs (van den Berg 2015).

3.4 Cloud delivery models

Another way to classify clouds is by their delivery models. There are three of them: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Figure 6 below compares these three cloud delivery models to traditional on-premises hosting by showing which components of a service are managed by a consumer and which ones by an external provider.

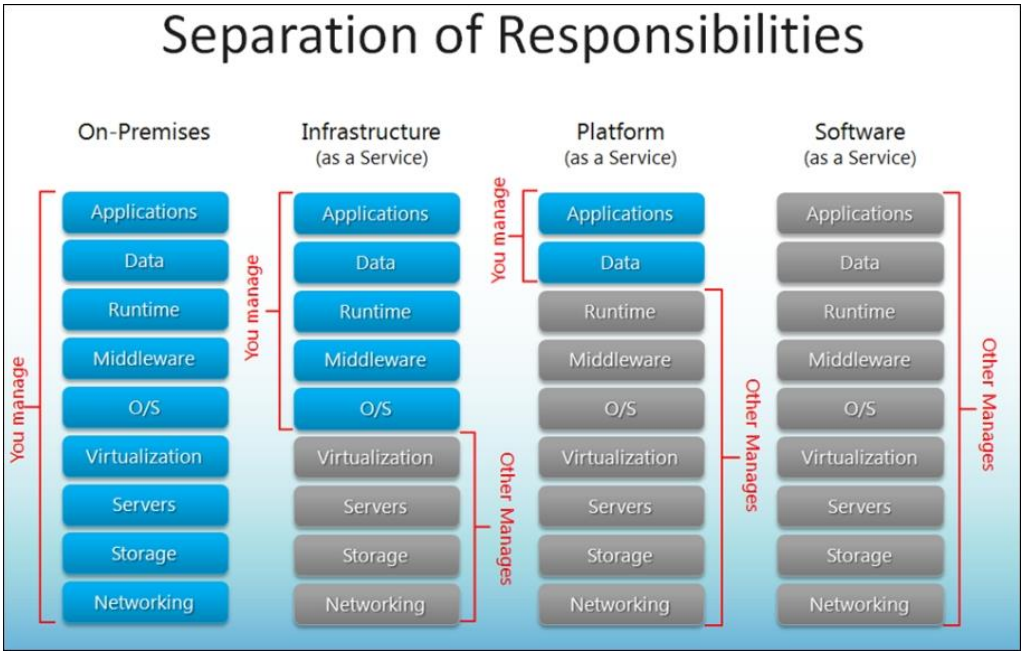

Figure 6. Separation of responsibilities (van den Berg 2015)

3.4.1 SaaS (Software as a Service)

In SaaS (Software as a Service) all the components of an application are hosted and managed by an external cloud provider. All the necessary for a service to run is installed on a provider's side and users can access a service over the Internet without a need to install something on their machines. Providers are responsible for keeping a service up and running, installing updates and managing underlying infrastructure. (Vadapalli 2017.)

SaaS is valued for low initial investment cost; no need to worry about hardware, software and licensing as it is managed by a provider; use of SSL (Secure Sockets Layer) that enhances security; low maintenance cost; fast setup, stability and reliability (Vadapalli 2017). The downside is that there is no control over the underlying infrastructure as it is managed by an eternal provider. In case of crash or downtime, a customer is not able to deliver its services anymore and must rely on a provider to fix the problems as soon as possible.

Software as a Service is usually used to deliver services over the Internet either for free or on subscription basis. Examples include Salesforce.com (and other customer relationship management platforms), Outlook 365, Hotmail, Yahoo! etc. These services are accessed over a web browser from anywhere and anytime. All the data, credentials, messages etc. are stored on a SaaS provider's side. (Microsoft Azure 2018.)

3.4.2 IaaS (Infrastructure as a Service)

Infrastructure as a Service delivery model provides virtualization, storage, networking and infrastructure itself. The rest of the components are managed by a customer. Consumers only pay for the infrastructure that is setup in the cloud and concentrate on what the service does rather than where it runs. A service can be managed over the Internet in a special customer's account. For example, creation of a virtual machine is a matter of a couple of clicks in a user-interface and is ready in a couple of minutes. (Khan et al 2017.)

Compared to SaaS, Infrastructure as a Service solution is cheaper because only infrastructure is rented. Consumer is responsible for managing the other components of a server. IaaS clouds are easily scalable because new virtual machines can be created in minutes. However, security and availability are still not controlled by a customer. That is why it is important to choose a reliable cloud provider. (Hindle 2015.)

SaaS is chosen if for a service to function properly developers need control over memory, an operating system or runtime. They can also build several redundant services to stay available in case of downtime. (Kavis 2014.) Most popular IaaS cloud providers are Microsoft Azure, Amazon Web Services, Google Compute Engine and IBM SmartCloud Enterprise.

3.4.3 PaaS (Platform as a Service)

Platform as a Service delivery model implies that a customer has control over an application and data while the rest is managed by an external cloud provider. This is an easy way for software developers to deploy their services on the cloud without planning the underlying infrastructure. (Goscinski et al. 2011.) There are several types of PaaS: Development as a Service, Database as a Service, Data as a Service and open source PaaS (Vadapalli 2017).

As it was already mentioned for the previous delivery models, saving on hardware is the main benefit of PaaS. Companies do not need to worry about maintaining infrastructure and can concentrate on developing their services. They are free in choosing operating systems, languages and databases. (Patankar 2015.) Lack of control and security problems are downsides of Platform as a Service.

PaaS is the least mature delivery model. At the beginning it was not widely used because it was not compatible with many languages. However, in the last couple of years new PaaS providers emerged offering support for PHP, Ruby, node.js etc. Platform as a Service solution is ideal for workflow-driven services. (Kavis 2014.) Some of PaaS providers are Google App Engine, Acquia Cloud and Force.com.

3.5 Cloud providers

When it comes to choosing a cloud provider it is important to find a company that guarantees constant availability, reliable infrastructure and flexible pricing models. Most of the biggest IT companies have their cloud computing platforms. Figure 7 shows which cloud providers dominated the market in 2016 and 2017.

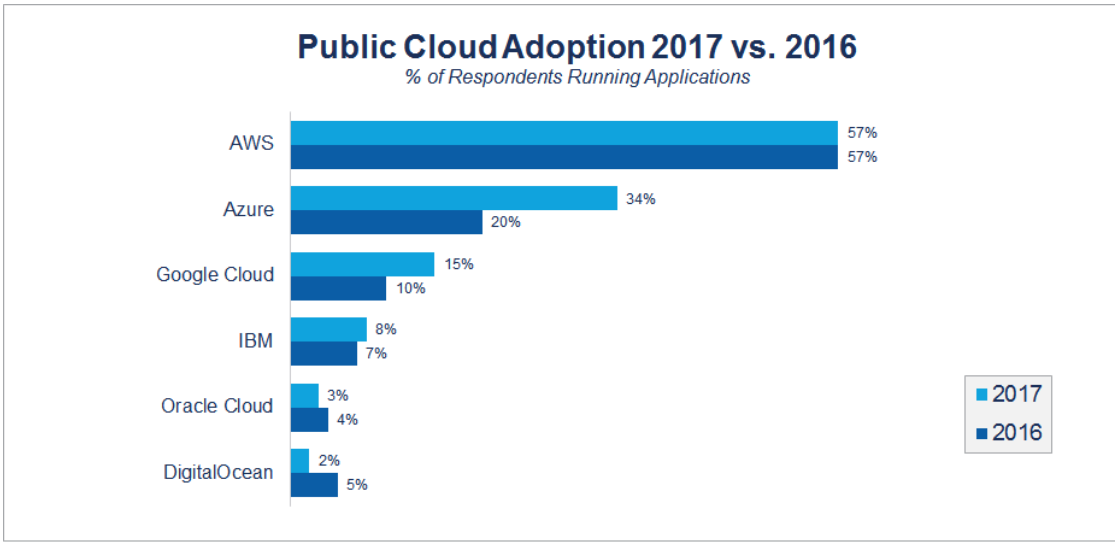

Figure 7. Public Cloud Adoption 2017 vs. 2016 (Kavis 2017)

Amazon Web Services is the leader of the industry, with 57% of respondents using its cloud services. Microsoft Azure and Google Cloud Platform are on the second and on the third places respectively. In 2017 they both gained more popularity as did IBM with a slightly smaller market share. Oracle Cloud and DigitalOcean are gradually losing their popularity. Three market leaders will be described in this section.

3.5.1 Amazon Web Services

Amazon Web Services, also called AWS, is the most popular cloud provider. It sells virtual machines, databases, RESTful APIs etc. Andrawos and Helmich (2017) list some of AWS services:

- Elastic Compute Cloud (EC2) is the most popular AWS service that povides capacity that is easy to manage and scale. It has a pay-as-you-go pricing model so that a customer only pays for the capacity that is actually used. EC2 is secure, reliable and easy to start. (Amazon Web Services 2018a.)

- Amazon S3 is a data storage service that is capable of storing, collecting and analyzing big volumes of data. It also includes query-in-place functionality and compliance capabilities. Amazon S3 is valued for its durability, flexibility and security. (Amazon Web Services 2018a.)
- Amazon DynamoDB is a NoSQL database that is delivered as a service. There are two store models: document and key-value. Dynamo DB is highly scalable, flexible and fully managed. (Amazon Web Services 2018a.)
- Amazon API Gateway provides an easy way for developers to create, manage and monitor APIs. At the same time, the application's back-end can be stored in EC2. Amazon API Gateway is rather cheap, easy to monitor, scalable and secure. (Amazon Web Services 2018a.)

The main advantage of AWS is its size. It has a network of data centers located in different parts of the world and it provides much bigger capacity than any other cloud provider. Of course, the cloud is not something infinite and customers' services will grow only to a certain limit. Therefore, scalability should always be planned in advance. (Linton 2011.)

AWS has some of the biggest corporations and governmental institutions among its clients, including the US federal government. Amazon was the first cloud provider to appear on the market in 2006 and has always been its leader. Many other providers appeared since then, but Amazon is the one that implements the newest technologies and serves as a reference point for the others. (Havanki 2017.)

3.5.2 Microsoft Azure

Microsoft describes its cloud solution as: "a growing collection of integrated cloud services ─ analytics, computing, database, mobile, networking, storage, and web ─ for moving faster, achieving more, and saving money" (Rabeler 2017). According to Andrawos and Helmich (2017), its services are similar to those provided by AWS with some slight differences:

- Virtual Machines – management of virtual machines
- Azure SQL – management of Microsoft SQL databases
- Azure Cosmos DB – management of NoSQL databases
- Queue Storage – management of storage
- Application Gateway – management of APIs (Andrawos & Helmich 2017.)

At the beginning, Azure supported only Microsoft native products and technologies, such as .NET, SQL Server, Windows, and was called Windows Azure. It was later renamed to Microsoft Azure to show the extended support of other technologies. As well as AWS, Microsoft has a wide network of data centers all over the world that ensure high availability. Another advantage is that during the first month of Azure subscription there is a $200 credit that can be used to explore different services and to understand whether it is suitable for a given project. (Hoffman 2018.) Unfortunately, Azure is also known for its complexity when it comes to management and even performing every-day tasks. That is why there is a certain learning curve that should be considered.

Since Microsoft has been on the market for a long time and has big corporations among its clients, many of them started to use Azure as a cloud provider. (Havanki 2017.) Nevertheless, it also works well with smaller players. For them it is a good opportunity to benefit from cloud computing and to save on infrastructure costs. (Hoffman 2018.)

3.5.3 Google Cloud Platform

The history of cloud solutions from Google started in 2008 when Google App Engine was released. It was a service for developers that facilitated the development of web applications. The next tools were Google Cloud Storage for managing capacity and BigQuery for database management. Google Cloud Platform appeared in 2013 and positioned itself as a competitor of Amazon’s EC2 service. (Havanki 2017.)

Again, being a big player on the IT market, Google boasts with one of the largest computer networks in the world. What is more, it owns a private fiber-optic cable that is located under the Pacific Ocean. Google’s data centers are efficient, reliable and modern. This results in a lower price for its customers. Google is constantly developing innovative software solutions and its clients are the first to try them out. (Ugia Gonzalez & Krishnan 2015.) However, while Google Cloud Platform works well with Google native technologies, the support for the others is limited. Google Cloud Platform does not have as

many services as, for example, AWS, but it is a good tool to start using the cloud.

4 TOOLS AND TECHNOLOGIES

Before starting any practical work, it is important to set up a development environment, decide which languages and frameworks to use and make sure a version control system keeps track on all the changes. All this will make that the workflow will go smoothly. This section describes some of the tools and technologies used for the practical part of this thesis.

4.1 Vagrant

Virtualization technologies have been used in web development for a long time. Virtual machines are an easy way to create a desired environment ─ whether it is Linux, Windows or any other operating system ─ regardless of the host OS. VMs are usually stored as images. However, virtual machines can be rather complicated to configure and to port (Peacock 2015). There are special tools to simplify the management of these VMs. One of them is Vagrant.

“Vagrant is a tool for building and managing virtual machine environments in a single workflow” (Vagrant 2018). It is a command line interface software that creates consistent environments that are easy to share between developers. To start a new virtual machine, it is enough to create Vagrantfile that specifies its configuration options. (Thompson 2015.) VirtualBox is used to create virtual machines themselves. They run on isolated environments, with the possibility to communicate with the host OS to some extent, as defined in Vagrantfile. Vagrant allows starting virtual machines in seconds.

This tool enhances cooperation inside a team of software developers and new members can be up and running with the project within minutes (Peacock 2015). What is more, developers are now sure that their environments are identical. System engineers will also be able to prepare development environment that are close to production. Vagrant is constantly gaining its popularity as an indispensable tool in web development. (Gajda 2015.)

4.2 API

While monolithic applications are still in use, it is now more common to break down a single web application into two logical parts – backend and frontend. Backend is usually implemented as an API. APIs (Application Programming Interface) provide data and services that can be reused for different purposes. Some of the most popular APIs are Google API, Facebook API or Twitter API. There are two types of APIs – public (that serve multiple clients) and private (used by companies or individuals). (Jacobson et al. 2011)

HTTP is currently the most widely used protocol for communication over the Internet. That is how applications communicate between each other. Client sends a request to an API using one of the request methods – GET, POST, PUT and DELETE. API then processes this request and sends back a response. (Simmons 2016.) It is usually sent in JSON format, that will be discusses later in this chapter.

Private API is a good starting point to modernize a legacy application and later deploy it to the cloud. Public APIs save time greatly as they decrease the amount of code needed to create an application and integrate it with, for example, Facebook or Google. While private API are created and managed for the purposes of a certain company or individual, public API are managed by external API providers. Thus, they can be shut down (for example, Google Health or Google Reader) and affect many applications that depend on them. (Proffitt 2013.) APIs is one of the standards in web development and whether they are public or private, they should definitely be considered while creating applications.

4.3 PHP

Backend (API) part can be written in different languages (among those .NET, PHP, Java etc.). PHP (full name PHP Hypertext Preprocessor) was chosen for this thesis as it is a server-side programming language that has been designed especially for web. PHP is also actively used for developing web services. It is estimated that most of the websites are running PHP on the back-end. (Thomson & Welling 2016.)

Plain PHP is not often used for developing complex web applications. When there is a need of controllers, models and views the best solution is to use one of the existing PHP frameworks or microframeworks. They help to keep the project organized and connect all its components (MVC, CSS, JavaScript). The microframework chosen for this thesis is Silex. It is easy to install, has all the basic features (router, request helpers etc.) and benefits from the library of service providers. It is designed to be lightweight and include only a minimum set of features and services. However, it is easily expandable. Silex, as well as its dependencies are installed through Composer. (Lopez 2016.) Silex is good for both monolithic applications and APIs.

4.4 JSON

According to Friesen (2016), “JSON (JavaScript Object Notation) is a language-independent data format that expresses JSON objects as human-readable lists of properties (name-value pairs).” JSON is typically used to return data in APIs or in AJAX communication. One of the reasons it became so popular is its simple and straightforward syntax. A JSON object is delimited with curly braces and separated by commas list of properties. JSON is programming language independent meaning that its syntax is recognized and supported by many languages. (Friesen 2016.)

JSON has become a de facto standard for a couple of reasons. There are more and more APIs based on it. Also, JavaScript is also becoming popular as a main development language, leading to a growing popularity of JSON. (Marrs 2017.) On the other hand, there are some downsides, among those no ability to add comments and no validation (Patni 2017). JSON is a core technology to learn to master APIs.

4.5 Version Control

Version control system is defined as: “a system capable of recording the changes made to a file or a set of files over a time period in such a way that it allows to get back in time from the future to recall a specific version of that file” (Somasundaram 2013). There are two main types of version control systems – centralized version control systems (CVCSs) and distributed version control systems (DVCSs). CVCS have a central server that hosts a repository as well

as all versions of files, tags and commits. They are done on local machines and pushed to the server. In DVCS, there is no need for a centralized server. Local copies of repositories include not only the files themselves, but also information about tags, versions and branches. In practice, a central repository still exists to represent official releases and forks. (Stachmann & Preißel 2014.)

Git is the most widely used distributed version control system. It was developed by Linus Torvalds in 2005 and became a de facto standard since then. Git is an open source software. (Voss et al. 2016.) It is a command-line tool that can be installed easily on any PC. It is usually used with GUI (graphical user interface) tools such as Source Tree or GitHub Desktop. A popular place to store repositories is GitHub (https://github.com).

5 IMPLEMENTATION

This chapter describes step by step the process of the migration of a legacy application to the cloud. First, an application was developed in a monolithic (legacy) way and was later converted into an API. It was then split into a group of Docker containers and deployed on Amazon Web Services. The full source code can be found on GitHub at https://github.com/yevinosyk/cloud-migration .

5.1 Creating a legacy web application

The reason why a cloud-native application was not built from the very beginning is that in a working-life situation there is usually no time and resources to develop new applications from the scratch. Instead, old ones are refactored to be suitable for containerization and the cloud. The same approach was chosen for this thesis. The idea was to create a simple Silex application where a user could add keywords and then link them to each other. For each of the keywords, it was possible to click on it and see its link. (van der Woude 2017.)

The first step is to set up the development environment. Both Vagrant and VirtualBox must be installed on the host PC. As it was already mentioned, Vagrantfile is used to declare what kind of a virtual machine should be created. Code 1 illustrates Vagrantfile that was added to the project directory.

A virtual machine called `cloudmigration` will be created from a `debian/stretch64` image with 1024MB of RAM and 2 CPUs. The dependencies to be installed are listed in Install all required software, Config PHP-FPM and NGINX and install composer sections. To start the virtual machine, a `vagrant up` command is run followed by `vagrant ssh` to start an SSH session.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.box = "debian/stretch64"

  config.vm.hostname = "cloudmigration.dev"
  config.vm.network "private_network", ip: "***.***.***.***"
  config.ssh.forward_agent = true

  config.vm.synced_folder ".", "/vagrant", type: "nfs"

  config.vm.provider "virtualbox" do |vb|
    vb.customize ["modifyvm", :id, "--memory", "1024"]
    vb.customize ["modifyvm", :id, "--cpus", "2"]
    vb.customize ["modifyvm", :id, "--name", "cloudmigration"]
  end

  # Install all required software
  config.vm.provision "shell", inline: "apt-get update; apt-get upgrade -y"
  config.vm.provision "shell", inline: "apt-get install -y nginx php-fpm php-zip
php-xml php-sqlite3"

  # Config PHP-FPM and NGINX
  config.vm.provision "shell", inline: "cp /vagrant/install/nginx_vhost_silex.conf
/etc/nginx/sites-enabled/default; cp /vagrant/install/php_fpm_pool.conf
/etc/php/7.0/fpm/pool.d/www.conf; service nginx restart; service php7.0-fpm restart"
  config.vm.provision "shell", inline: "systemctl enable php7.0-fpm ; systemctl
enable nginx"

  # Install composer
  config.vm.provision "shell", inline: "cd /var/tmp && wget
https://getcomposer.org/download/1.5.6/composer.phar && mv composer.phar
/usr/bin/composer && chmod 755 /usr/bin/composer"
end
```

Code 1. Vagrantfile (van der Woude 2017)

A GitHub repository called yevinosyk/cloud-migration was created to host all the project files. It was cloned to the local machine with the `git clone https://github.com/yevinosyk/cloud-migration.git` command. The commits were made locally and then pushed to GitHub with `git push origin master`.

In the virtual machine, the applications will be placed under the following directories: `/vagrant/APP_ROOT/silex_web` for the legacy application and `/vagrant/APP_ROOT/silex_api` for the refactored application.

To set up the first Silex application, a `composer require silex/silex "~2.0"` command is used. "Composer is a dependency manager application that can be used to install PHP packages." It makes it easy to generate updates and keep track on all the dependencies that are installed in each project. The dependencies are taken from a package repository called Packagist. (Salehi 2016.) The information about the packages used is added to `composer.json` and `composer.lock` files (located in the project directory).

The application directory (`/silex_web`) has the following structure:

- src – controllers, models and services (also referred to as bundles);
- vendor – third-party plugins and packages;
- views – templates;
- web – main application files (`app.php`), stylesheets and JavaScript, all of them are accessible from the web. (Bancer 2015.)

To begin, the following dependencies will be installed using Composer and then registered as services in the `app.php` file (Code 2):

- Doctrine is an object relational mapper (ORM) that enables database management directly in PHP code (using PHP classes and methods). SQLite is a lightweight database that will be configured using Doctrine. (Salehi 2016.)
- Twig is a template engine that generates views. Originally PHP was used to develop views. However, it made them too complex. Twig's syntax is simpler. (Bancer 2015.)
- Asset is a component that handles CSS and JavaScript files (assets) that are placed under /web directory. Links to them are placed in templates.
- Form is a component that manages form creation, processing and validation (Bancer 2015).

```
$app->register(new DoctrineServiceProvider(), array (
    'db.options' => array(
        'driver' => 'pdo_sqlite',
        'path' => '/var/tmp/db'
    ),
));

$app->register(new Silex\Provider\AssetServiceProvider(), array(
    'assets.named_packages' => array(
        'css' => array('version' => 'css3', 'base_path' => '/css'),
        'images' => array('base_path' => '/img'),
    ),
));

$app->register(new FormServiceProvider());

$app->register(
    new TwigServiceProvider(),
    ['twig.path' => __DIR__ . '/../views']
);
```

Code 2. Services registered in app.php

As it was already mentioned, all the templates will be stored in the `/views` directory. `Layout.twig` includes basic html tags (html, head, title, body) and links to stylesheets (Code 3). This code is common for all the templates. That is why it is written only once in `layout.twig` and will be reused by all the other templates as they will be its extensions.

```
<html>
    <head>
        <title>Keywords</title>
        {% block stylesheets %}
            <link href = "{{ asset('/css/bootstrap.css') }}" rel="stylesheet">
            <link href = "{{ asset('/css/styles.css') }}" rel="stylesheet">
        {% endblock %}
    </head>
    <body>
    {% block content %}
    {% endblock %}
    </body>
</html>
```

Code 3. Layout.twig template

The application's logic will be defined in a controller located in `/src/Controllers/NodesController.php`. It is called when a certain route is matched (Silex 2018). All the routes are added to `app.php` (Code 4). Depending on the URL entered, a certain method from the controller will be executed.

```
$app->get('/','keywords.controller:getAll');
$app->get('/node/{id}','keywords.controller:getNode');
$app->post('/keywords','Keywords\\Controllers\\NodesController::create');
$app->get('/keywords','Keywords\\Controllers\\NodesController::getNewForm');
$app-
>get('/node/{id}/create_link','Keywords\\Controllers\\NodesController::newLinkForm')
;
$app-
>post('/node/{id}/create_link','Keywords\\Controllers\\NodesController::createLink')
;
```

Code 4. Routes in app.php

The first route (/) invokes a `getAll()` method (Code 5). First, it checks that the database is created using a private `checkSchema()` function (Code 6). Then, it runs a query that retrieves all the values from nodes table and adds them to a `nodes` array. They are finally added to a `get_all.twig` template that displays all the keywords that have already been added.

```
public function getAll(Application $app): string {
     $this->checkSchema($app);

     $sql = 'SELECT * FROM nodes';
     $nodes = $app['db']->fetchAll($sql);
     return $app['twig']->render('get_all.twig', array('nodes' => $nodes));
}
```

Code 5. getAll method in NodesController.php (van der Woude 2018)

```
private function checkSchema($app)
{
    $schemaResult = $app['db']->fetchAll('PRAGMA table_info(nodes)');

    if (empty($schemaResult)) {
        $schema = [
          'CREATE TABLE nodes(id INTEGER PRIMARY KEY NOT NULL, node TEXT NOT NULL)',
          'CREATE TABLE links(id INTEGER PRIMARY KEY NOT NULL, node_1 INTEGER NOT
           NULL, node_2 INTEGER NOT NULL, FOREIGN KEY(node_1) REFERENCES nodes(id),
           FOREIGN KEY(node_2) REFERENCES nodes(id))'
         ];
        foreach ($schema as $sql) {
            $app['db']->executeUpdate($sql);
        }
      } else {
    }
}
```

Code 6. checkSchema method in NodesController.php (van der Woude 2018)

The second route (`/keywords`) invokes two methods: `create()` (Code 7) and `getNewForm()` (Code 8). These methods check whether the database exists and renders a form created using `getForm()` method (Code 9). This form allows a user to add a new keyword. If the form is valid, it checks whether the keyword entered already exists using the `getNodeByName()` method that retrieves the names of nodes from the database, as shown in Code 10. If it is the case, a form error message will be displayed. If a new keyword is not yet in the database, it will be added to the nodes table and at the end the user will be redirected to the home page (/) that lists all the added keywords. Finally, a `new_keyword.twig` template provides a user interface for all the actions mentioned above.

```
public function create(Application $app, Request $request): string {
    $this->checkSchema($app);
    $form = $this->getForm($app);
    $form->handleRequest($request);

    if ($form->isValid()) {
        $data = $form->getData();
        if ($this->getNodeByName($app, $data['node'])) {
            $form->addError(new FormError('That node already exists'));
            }
        }

    if ($form->isValid()) {
        $data = $form->getData();
        $sql = 'INSERT INTO nodes (node) VALUES (:node)';
        $result = $app['db']->executeUpdate($sql, array('node' => $data['node']));
        return $app->redirect('/');
    }

return $app['twig']->render('new_keyword.twig', array('form' => $form-
>createView()));
}
```

Code 7. A create method in NodesController.php (van der Woude 2018)

```
public function getNewForm(Application $app): string {
    $form = $this->getForm($app);

    return $app['twig']->render('new_keyword.twig', array('form' => $form-
>createView()));
}
```

Code 8. getNewForm method in NodesController.php (van der Woude 2018)

```
private function getForm($app) {
    return $form = $app['form.factory']->createBuilder(FormType::class, null)
        ->add('node', null, array(
            'label' => false,
            'attr' => array('class'=>'form-control form-control-lg')
        ))
        ->getForm();
}
```

Code 9. getForm method in NodesController.php (van der Woude 2018)

```
private function getNodeByName($app, $node)
{
    $sql = 'SELECT * FROM nodes WHERE lower(node)=:node';

    $result = $app['db']->fetchAll($sql, array('node' => strtolower($node)));
    if (count($result) > 0) {
        return $result[0];
    }
    return null;
}
```

Code 10. getNodeByName method in NodesController.php (van der Woude 2018)

The `get_all.twig` template lists all the keywords and shows a “Link it!” button newt to them. This button is a link for the `/node/{id}/create_link` route. It invokes two methods: `newLinkForm()` (Code 11) and `createLink()` (Code 15). First, the `newLinkForm()` method fetches the id of the keyword that is being linked using a `getNodeById()` method (Code 12). A form for creating a link is generated with a `getLinkForm()` method (Code 13). The form itself consists of a drop down that displays all the keywords, except for the one that is been linked. It also validates that the same link does not exist yet, using a `getNodeLinks()` method (Code 14). At the end, a `new_link.twig` template is rendered to display the form to a user. The `createLink()`method (Code 15) handles the creation of a link and adds it to the `links` table in the database.

```
public function newLinkForm(Application $app, $id): string {
    $node = $this->getNodeById($app, $id);

    if (!$node) {
        throw new \Exception('The node you are trying to link to does not exist');
    }

    $form = $this->getLinkForm($app, $node);
    return $app['twig']->render('new_link.twig', array('form' => $form-
>createView(), 'node' => $node));
 }
```

Code 11. newLinkForm method in NodesController.php (van der Woude 2018)

```
private function getNodeById($app, $id)
{
    $sql = 'SELECT * FROM nodes WHERE id=:id';
    $result = $app['db']->fetchAll($sql, array('id' =>$id));

    if (count($result) > 0) {
        return $result[0];
    }
    return null;
}
```

Code 12. getNodeById method in NodesController.php (van der Woude 2018)

```
private function getLinkForm($app, $node) {
$sql = 'SELECT * FROM nodes WHERE id != :id';

$nodes = $app['db']->fetchAll($sql, array('id' => $node['id']));

$choices = array();
foreach ($nodes as $n) {
$choices[$n['node']] = $n['id'];
}

$links = $this->getNodeLinks($app, $node);
$data = array();
foreach ($links as $link) {
$data[] = $link['id'];
}

return $form = $app['form.factory']->createBuilder(FormType::class, null)
                ->add('links', ChoiceType::class, array(
                    'label' => false,
                    'choices' => $choices,
                    'multiple' => true,
                    'data' => $data
                ))
                ->getForm();
}
```

Code 13. getLinkForm method in NodesController.php (van der Woude 2018)

```
private function getNodeLinks($app, $node)
{
    $sql = 'SELECT n.id,n.node FROM links l LEFT JOIN nodes n on n.id=l.node_1 WHERE
l.node_2=:id';

    $nodes = $app['db']->fetchAll($sql, array('id' => $node['id']));
    return $nodes;
}
```

Code 14. getNodeLinks method in NodesController.php (van der Woude 2018)

```
public function createLink(Application $app, Request $request, $id): string {
    $this->checkSchema($app);

    $node = $this->getNodeById($app, $id);
    if (!$node) {
        throw new \Exception('The node you are trying to link to does not exist');
    }

    $existingLinks = $this->getNodeLinks($app, $node);
    $form = $this->getLinkForm($app, $node);

    $form->handleRequest($request);

    if ($form->isValid()){
```

```
        $data = $form->getData();

        foreach ($data['links'] as $linkId) {
            $found = false;
            foreach ($existingLinks as $link) {
                if ($link['id'] == $linkId) {
                    $found = true;
                }
            }
            if (!$found) {
                $sql = 'INSERT INTO links VALUES (null, :node_1, :node_2)';
                $result = $app['db']->executeUpdate($sql, array('node_1' => $linkId,
'node_2' => $id));
                $result = $app['db']->executeUpdate($sql, array('node_1' => $id,
'node_2' => $linkId));
            }
            }
            foreach ($existingLinks as $link) {
                $found = false;
                    foreach ($data['links'] as $linkId) {
                        if ($link['id'] == $linkId) {
                            $found = true;
                        }
                    }

                    if (!$found) {
                        $sql = 'DELETE FROM links WHERE (node_1=:id and
node_2=:linkid) OR (node_1=:linkid and node_2=:id)';
                        $result = $app['db']->executeUpdate($sql, array('id' => $id,
'linkid' => $link['id']));
                    }
                }
        return $app->redirect('/');
    }
return $app['twig']->render('new_link.twig', array('form' => $form->createView()));
        }
```

Code 15. createLink method in NodesController.php (van der Woude 2018)

The last route defined in `app.php` is `/node/{id}` that invokes a `getNode` method (Code 16). It returns a `get_node.twig` template when a keyword is clicked that shows the keyword itself and all its links.

```
public function getNode(Application $app, $id): string {
    $node = $this->getNodeById($app, $id);

    if (!$node) {
        throw new \Exception('this is not the node you are looking for');
    }

    $links = $this->getNodeLinks($app, $node);
    return $app['twig']->render('get_node.twig', array('node' => $node, 'links' =>
$links));
}
```

Code 16. getNode method in NodesController.php (van der Woude 2018)

The final result can be seen in Figure 8.

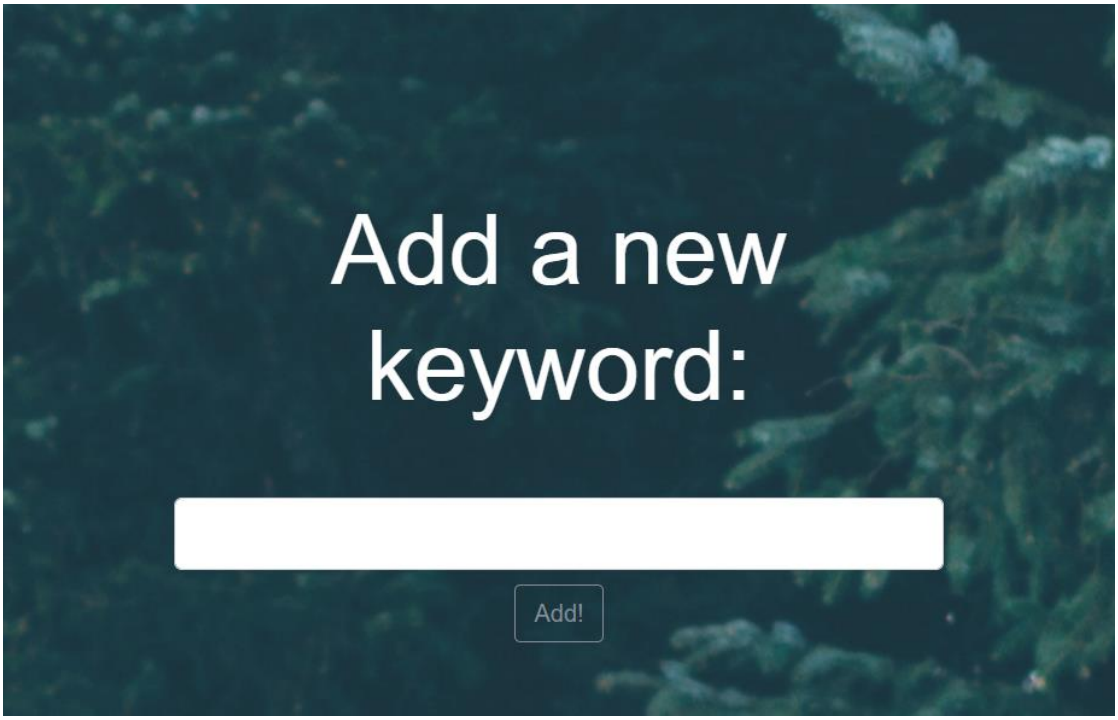

Figure 8. Ready application

A simple Silex application has just been created. The idea was to be able to add keywords and later link them with the other keywords. Now it is a monolithic (legacy) application and it will be modified in the following chapter, so that it can later be deployed on the cloud.

5.2 Creating an API

The monolithic application created above is a perfectly working example. However, nowadays it is more common to create APIs (Application Programming Interfaces). They return data in JSON format that can later be used by, for example, frontend frameworks or other applications.

First, all the project files will be copied from `/APP_ROOT/silex_web/` to `/APP_ROOT/silex_api/` so that both the legacy application and the API are saved. The next step is to modify the Silex application so that it becomes an API and returns data in JSON format. All the frontend components must be removed, because an API represents the backend of the application. Twig and assets will be deleted using `composer remove <package_name>`. CSS files and templates, located in `/APP_ROOT/silex_api/web/css`, `/APP_ROOT/silex_api/web/img` and `/APP_ROOT/silex_api/views` will be

deleted manually. The declarations of service providers, as shown in Code 2, are also to be deleted.

The methods defined in `NodesController.php` should be modified so that they return data in JSON format rather than render it in the templates. “The `JsonResponse` class sets the Content-Type header to `application/json` and encodes data to JSON when needed” (Symfony 2018). Code 17 illustrates the use of this class in the `getAll()` method. The methods `newLinkForm()` and `getNewForm()` are deleted, because forms will not be used in the API. The other methods to return JsonResponse are `getNode()` (Code18) and `postNode()` (Code 19).

```
public function getAll(Application $app): string {
    $this->checkSchema($app);

    $sql = 'SELECT * FROM nodes';
    $nodes = $app['db']->fetchAll($sql);

    return new JsonResponse($nodes);
}
```

Code 17. getAll() method returns data as JsonResponse

```
public function getNode(Application $app, $id): string {
    $node = $this->getNodeById($app, $id);

    if (!$node) {
        throw new \Exception('this is not the node you are looking for');
    }

    $node['links'] = $this->getNodeLinks($app, $node);
    return new JsonResponse($node);
}
```

Code 18. getNode() method returns data as JsonResponse

```
public function postNode(Application $app, Request $request): string {
    $this->checkSchema($app);

    $form = $this->getForm($app);

    $form->handleRequest($request);

    if ($form->isValid()) {
        $data = $form->getData();
        if ($this->getNodeByName($app, $data['node'])) {
            $form->addError(new FormError('That node already exists'));
        }
    }

    if ($form->isValid()) {
        $data = $form->getData();

        $sql = 'INSERT INTO nodes (node) VALUES (:node)';
        $result = $app['db']->executeUpdate($sql, array('node' => $data['node']));
        return $app->redirect('/');
    }
```

```
    return new JsonResponse($form->getErrors(true, true));
}
```

Code 19. postNode() method returns data as JsonResponse

Now when the Silex application has been converted into an API, it returns JSON data in the browser, as can be seen in Figure 9.

HTTP/1.0 200 OK Cache-Control: no-cache, private Content-Type: application/json Date: Sat, 03 Feb 2018 11:27:34 GMT [{"id":"1","node":"Munich"},{"id":"2","node":"chicken"}, {"id":"3","node":"curry"},{"id":"4","node":"thesis"},{"id":"5","node":"containers"}, {"id":"6","node":"cloud"}]

Figure 9. JSON response returned in the browser

5.3 Containerizing the application

In this part, the previously created application will be divided into two logical parts that will then be deployed to Docker containers. The first thing to do is to install Docker. Code 20 shows the commands to set up a Docker repository. It updates existing packages, installs the ones needed for Docker, adds Docker's GPG key and sets up a stable repository. (Docker 2018.)

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates curl
   gnupg2 software-properties-common
$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/debian
  $(lsb_release -cs) stable"
```

Code 20. Setting up Docker repository (Docker 2018)

Docker comes in two editions – Community and Enterprise. For this project, The Community Edition will be installed. Code 21 shows the commands to download the Docker package and check that it is correctly installed by running a simple `hello-world` test.

```
$ sudo apt-get update
$ sudo apt-get install docker-ce
$ sudo docker run hello-world
```

Code 21. Installing Docker Community Edition (Docker 2018)

The application will be divided into two containers – `php-fpm` container and `nginx` container. These containers can either be started manually every time

they are needed, or this process can be automated. Docker Compose is used to handle multi-container applications. It stores information about all the containers in a YAML file called `docker-compose.yml` that is then started with a single command. (McKendrick & Gallagher 2017.) Code 22 shows the commands to install Docker Compose.

```
$ sudo curl -L https://github.com/docker/compose/releases/download/1.18.0/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

$ sudo chmod +x /usr/local/bin/docker-compose
```

Code 22. Installing Docker Compose (Docker 2018)

The Docker Compose file will be created in `/vagrant/docker/docker-compose.yml`. Code 23 shows how this file is organised. It defines containers as services and each of the services will later include information about image, links, mounted directories and ports.

```
version: '3.4'
  services:
    #nginx container
    web:
    #php-fpm container
    php:
```

Code 23. Docker-compose.yml file's structure

Docker containers start with an image. It is a file that contains an operating system, dependencies and the application itself. Docker Hub is an official registry of Docker images. They can be either official or public. (Stoneman 2017.) While official images are considered safe to use, public images must be used with caution as there is no guarantee that they are secure. Docker Hub has the `nginx` and `php:7.0-fpm` official images, and they will be used for this project. However, these images still need some customization to run certain commands and mount volumes.

Dockerfile is used when there is a need to create a customised Docker image. The following directories should be created: `/docker/nginx` and `/docker/php-fpm`. The Dockerfile for pfp-fpm container will be placed in `/docker/php-fpm/Dockerfile`. Its content is shown in Code 24. This Dockerfile created a new container based on the `php:7.0-fpm` image from Docker Hub. It also runs bash commands to update packages and install missing ones, especially important is curl. Next, composer is installed. The copy of the `/silex_api/composer.json` file is added to /`docker/php-`

`fpm/composer.json` so that dependencies listed there can be installed on the container. Usually, a database would require a separate container. But there is no separate database process, because Silex uses `pdo_sqlite` provider that writes to `/var/tmp/db`.

```
FROM php:7.0-fpm

EXPOSE 9000

RUN apt-get update && apt-get install -y --no-install-recommends \
    curl \
    git \
    sudo \

RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin
--filename=composer

WORKDIR /usr/src/silex

ADD ./composer.json /usr/src/silex

RUN composer install --prefer-dist --working-dir=/usr/src/silex

VOLUME ./../silex_api
```

Code 23. Dockerfile for php-fpm container

As for the nginx image, the public one works well. The default nginx configuration is copied to `/docker/nginx/default.conf`. Code 24 illustrates the ready `docker-compose.yml file`. It builds a `php` container from the `Dockerfile` and `nginx` container directly from the image.

```
version: '3.4'
services:
  php:
    build: ./php-fpm
    volumes:
      - ./../silex_api:/usr/src/silex/app
  nginx:
    image: nginx
    links:
      - php
    ports:
      - "8080:80"
    volumes:
      - ./../silex_api:/usr/src/silex/app
      - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
```

Code 24. Docker-compose.yml file

Containers are run with `sudo docker-compose up -d` in the same directory as the `docker-compose.yml` file and stopped with `sudo docker-compose stop`.

5.4 Moving the application to the cloud

Docker containers are ready, and they can be deployed on the Amazon Web Services cloud provider. It has a service called Amazon Elastic Container Service (ECS) that “is a highly scalable, high-performance container orchestration service that supports Docker containers and allows to easily run and scale containerized applications on AWS” (Amazon Web Services 2018b). First, a free AWS account should be created. To start using Elastic Container Service, some additional setup is necessary. At this stage, configuration is done on the AWS website.

For security reasons, it is recommended to access AWS with a root user account. It is done at https://console.aws.amazon.com/iam/ . The user Administrator and group Administrators are created. To log in as a root user, the URL is in the following format: https://your_aws_account_id. signin.aws. amazon.com/console/. (AWS 2018.) AWS is also available as a command line tool. Code 25 shows commands necessary to install it.

```
$ sudo apt-get install python-pip
$ sudo pip install awscli
```

Code 25. Installing AWS CLI

Before the local Docker images will be pushed to AWS, a repository that will host them is needed. It is created following the Repositories wizard. When its name is chosen (thesis-project), images can be pushed to this repository via the command line interface. Code 26 shows the necessary commands. First, authentication is established. Then, images are built (two images were created in the previous chapter: docker_php and nginx). When Images are successfully built, they can be tagged and pushed.

```
$ sudo aws ecr get-login --no-include-email --region us-east-2
$ sudo docker build -t <image_name> .
$ sudo docker tag <image_name>:latest <account_id>.dkr.ecr.us-west-
2.amazonaws.com/thesis-project:latest
$ sudo docker push <account_id>.dkr.ecr.us-west-2.amazonaws.com/ thesis-
project:latest
```

The next step is to create a new task definition. It describes which containers to run, from which images etc. It is also configured in a wizard called Task Definitions. As it can be seen, the AWS graphical user interface is used to

handle most of the configuration. Specifying a new task's name (task) and at least one container is enough to create it. There are additional, but not required options, such as Task Size (memory and CPU) or Task Role. Containers are then added by specifying the repository URI that has the image needed and giving the container a name. When containers are added and configured, a new Task Definition can be created.

Amazon ECS Clusters are used to logically group task definitions. They are created in the Clusters wizard. To create a new cluster, the following required parameters are specified: name, instance type (t2.micro is chosen, it has 1 vCPU and 1GiB of memory) and number of instances (1). The new cluster is then created. The next step is to define a new service. It will specify how many copies of the task definition should be run (1). When the service is up and running, it means that containers are successfully running as well.

To sum up, Amazon Web Services is an extremely powerful tool that offers a lot of services to its customers. The documentation describes how some typical tasks should be accomplished. Still, doing configurating in a graphical user interface is rather frustrating. It would be a good idea to add more documentation regarding the use of command line interface to accomplish similar tasks.

6 THE FUTURE OF CLOUD COMPUTING

It can be seen from the history of any technology that it is constantly evolving. Cloud computing is a logical result of the evolution of Service Oriented Architecture (SOA) and will continue its development in future. (Longbottom 2017.) The cloud has completely changed the direction in which information technology is moving. Figure 9 shows that it is estimated that the cloud will have gained even more popularity until 2020 resulting in nearly 40% of data stored or related to the cloud. However, it will still undergo some changes and challenges.

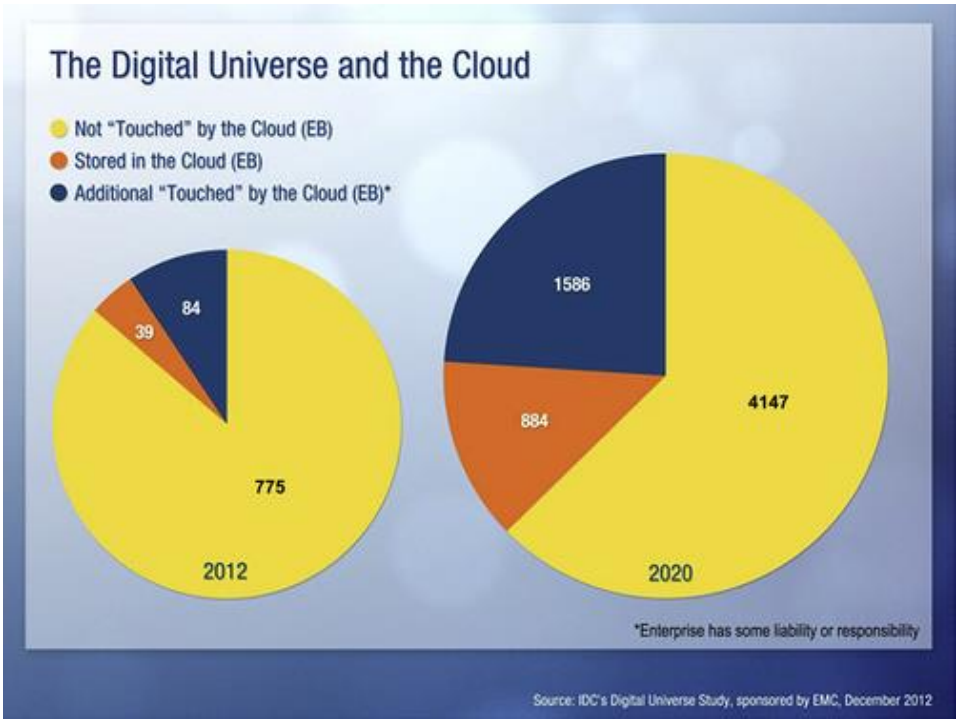

Figure 9. Growing role of cloud computing (Honan et al. 2014)

Nowadays, cloud computing is not the future but already the present. Companies see that the cloud is here and there is no other way but to take it into account in business operations. It is estimated that more and more enterprises will move their businesses to the cloud. The reality is that there is still a lot of resistance inside the companies themselves. Employees feel that they do not have enough expertise to keep up with the rapid changes of the IT industry. (Muikkunen 2018.) Among numerous cloud solutions available, companies must choose the ones that suit their requirements best. They will be able to deliver their services faster and cheaper than before, thus increasing the overall cloud adoption. (Kavis 2014.)

When cloud computing became widely used, companies shut down their data centers and stopped using their own hardware. Instead, they benefited from renting infrastructure from the cloud providers. However, they are now going even further and actively looking for ways to implement “serverless” architecture, also called Function-as-a-Service. Serverless means that cloud instances will no longer be allocated and sit idle until a specific event occurs.” This will result in cost savings, as customers will not pay for capacities when their services are idle. (Boulton 2018.) This concept is implemented by AWS Lambda.

Finally, cloud computing will continue developing its current trends. Software will be more and more separated from hardware and be delivered as a service. It will also become modular so that each part of software can be modified without affecting the whole program. Cloud service providers will benefit most from cloud computing as their services will have even higher demand. (Clark 2012.) Cloud computing has become so widespread that it is likely to affect technology as a whole, not only IT.

7 CONCLUSIONS

Cloud computing is playing a leading role in the development of the information technology industry. While a couple of years ago it was seen as a future of IT, it is now its reality that cannot be neglected. The knowledge of the cloud is necessary for anyone working in the IT industry.

The aim of this thesis was to study the area of cloud computing and see how it can be implemented in practice. The theory part described microservices and the cloud. It was important to look back in the history to see why these technologies evolved and which problems they solved. Their advantages and disadvantages were also discussed as well as ways to implement them in practice.

The idea for the practical part was to deploy a legacy web application to containers and then to the cloud. However, to accomplish this task, the knowledge only of the cloud is not enough. Additional tools are needed to create the application and make it cloud native. They were all discussed in a separate chapter. The implementation chapter described in a step by step manner what was done. First, a legacy web application was created. It was then converted into API. The application was split into Docker containers that were later deployed on Amazon Web Services.

Creating a legacy web application may seem as an unnecessary step. However, that is the way such problems are solved in real companies. There are rarely enough time and resources to build cloud native applications from the scratch. Instead, the existing ones must be refactored in a way that they

can be deployed to the cloud as quickly as possible. In this case, companies will benefit immediately from cloud computing.

It is difficult to overestimate the role of the commissioner company ─ NeosIT Services GmbH. The company made sure that the topic was up-to-date, and the implementation part was planned correctly. The company supervisor has a lot of experience in the field and gave valuable advice on how the work should be organized and which software and tools are the best ones to use. Seeing how real companies handle tasks is an invaluable experience for the future career.

The area of cloud computing is extremely extensive, and all the aspects cannot be covered in a single thesis. Nevertheless, the aim was achieved. The most popular cloud technologies were studied and used in practice.

REFERENCES

Amazon. 2018. Explore Our Products. WWW document. Available at: https://aws.amazon.com [Accessed 24 January 2018].

Amazon Web Services. 2018a. Amazon Elastic Container Service. WWW document. Available at: https://aws.amazon.com/ecs/ [Accessed 5 February 2018].

Amazon Web Services. 2018b. Setting Up with Amazon ECS. WWW document. Available at: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/get-set-up-for-amazon-ecs.html [Accessed 5 February 2018].

Andrawos, M., Helmich, M. 2017. Cloud Native programming with Golang. Birmingham: Packt Publishing.

Bancer, W. 2015. Symfony2 Essentials. Birmingham: Packt Publishing.

Bigelow, S. 2018. Five cons of container technology. WWW document. Available at http://searchservervirtualization.techtarget.com/feature/Five-cons-of-container-technology [Accessed 10 January 2018].

Boulton, C. 2018. Serverless: The future of cloud computing? WWW document. Available at https://www.cio.com/article/3244644/cloud-computing/serverless-the-future-of-cloud-computing.html [Accessed 10 January 2018].

Burns, B. 2017. Designing Distributed Systems. Sebastopol: O'Reilly Media, Inc.

Clark, J. 2012. Cloud computing: 10 ways it will change by 2020. WWW document. Updated 31 July 2012. Available at: http://www.zdnet.com/article/cloud-computing-10-ways-it-will-change-by-2020/ [Accessed 4 February 2018].

CloudAcademy. 2018. Why learn Cloud Computing? WWW document. Available at: https://cloudacademy.com/why-learning-cloud-computing/ [Accessed 10 January 2018].

Columbus, L. 2017. Roundup Of Cloud Computing Forecasts 2017. WWW document. Updated 29 April 2017. Available at: https://www.forbes.com/sites/louiscolumbus/2017/04/29/roundup-of-cloud-computing-forecasts-2017/#3dae3fb631e8 [Accessed 10 January 2018].

Dufficy, T. 2014. What is Private Cloud? Advantages and Disadvantages. WWW document. Updated 22 October 2014. Available at: http://www.serverspace.co.uk/blog/what-is-private-cloud-plus-advantages-disadvantages [Accessed 22 January 2018].

Docker. 2018. Get Docker CE for Debian. WWW document. Available at: https://docs.docker.com/install/linux/docker-ce/debian/ [Accessed 4 February 2018].

Docker. 2018. Install Docker Compose. WWW document. Available at: https://docs.docker.com/compose/install/ [Accessed 4 February 2018].

Fowler, M. 2014. Microservices. WWW document. Updated 12 January 2017. Available at: https://martinfowler.com/articles/microservices.html [Accessed 12 December 2017].

Friesen, J. 2016. Java XML and JSON. New York City: Apress.

Gajda, W. 2015. Pro Vagrant. New York City: Apress.

Goetsch, K. 2014. eCommerce in the Cloud. Sebastopol: O'Reilly Media, Inc.

Golub, B. 2017. DockerCon 2017 - General Session Day 1. WWW document. Updated 24 April 2017. Available at: https://www.slideshare.net/Docker/dockercon-2017-general-session-day-1-ben-golub [Accessed 21 January 2018].

Google Cloud Platform. 2018. Containers at Google. WWW document. Available at: https://cloud.google.com/containers/ [Accessed 18 January 2018].

Goscinski, A., Broberg, J., Buyya, R. 2011. Cloud Computing: Principles and Paradigms. Hoboken: John Wiley & Sons.

Havanki, B. 2017. Moving Hadoop to the Cloud. Sebastopol: O'Reilly Media, Inc.

Hindle, D. 2015. Infrastructure as a Service (IaaS) explained. WWW document. Updated 1 April 2015. Available at: https://www.computelec.com.au/infrastructure-as-a-service-iaas-explained/ [Accessed 23 January 2018].

Hoffman, C. 2018. What Is Microsoft Azure, Anyway? WWW document. Updated 4 January 2018. Available at: https://www.howtogeek.com/337961/what-is-microsoft-azure/ [Accessed 24 January 2018].

Honan, B., Reavis, J., Samani, R. 2014. CSA Guide to Cloud Computing. Rockland: Syngress.

Ivanov, K. 2017. Containerization with LXC. Birmingham: Packt Publishing.

Jacobson, D., Brail, G., Woods, D. 2011. APIs: A Strategy Guide. Sebastopol: O'Reilly Media, Inc.

Jain, H. 2016. LXC and LXD: Explaining Linux Containers. WWW document. Updated 2 June 2016. Available at: https://www.sumologic.com/blog/code/lxc-lxd-explaining-linux-containers/ [Accessed 21 January 2018].

Kavis, M. 2014. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS). Hoboken: John Wiley & Sons.

Kavis, M. 2017. Rightscale State of the Cloud Report 2017. WWW document. Updated 15 February 2017. Available at: https://www.virtualizationpractice.com/rightscale-state-cloud-report-2017-39791/ [Accessed 24 January 2018].

Kumina. 2017. Top 7 benefits of using containers. WWW document. Available at: https://blog.kumina.nl/2017/04/the-benefits-of-containers-and-container-technology/ [Accessed 18 January 2018].

Khan, O., Qureshi, H., Senthilvel, G. 2017. Enterprise Application Architecture with .NET Core. Birmingham: Packt Publishing.

Krochmalski, J. 2017. Docker and Kubernetes for Java Developers. Birmingham: Packt Publishing.

Lewish, J., Fowler, M. 2014. Microservices in a Nutshell. WWW document. Updated 27 June 2014. Available at: https://www.thoughtworks.com/insights/blog/microservices-nutshell [Accessed 4 December 2017].

Li, Y., Wu, L., Liu, S., Shen, Y., Wen, Q. 2013. Enabling the New Era of Cloud Computing. Hershey: IGI Global.

Linthicum, D. 2018. You've heard the benefits of containers, now understand the challenges. WWW document. Available at: https://techbeacon.com/youve-heard-benefits-containers-now-understand-challenges [Accessed 20 January 2018].

Linton, R. 2011. Amazon Web Services: Migrating your .NET Enterprise Application. Birmingham: Packt Publishing.

Longbottom, C. 2017. The Evolution of Cloud Computing: How to plan for change. Swindon: BCS Learning & Development Limited.

Lopez, A. 2016. Learning PHP 7. Birmingham: Packt Publishing.

Mahmood, Z., Erl, T., Puttini, R. 2013. Cloud Computing: Concepts, Technology & Architecture. Upper Saddle River: Prentice Hall.

Marrs, T. 2017. JSON at Work. Sebastopol: O'Reilly Media, Inc.

McKendrick, R., Gallagher, S. 2017. Mastering Docker - Second Edition. Birmingham: Packt Publishing.

Microsoft Azure. 2018. What is SaaS? WWW document. Available at: https://azure.microsoft.com/en-in/overview/what-is-saas/ [Accessed 23 January 2018].

Medium. 2016. Introduction to Containers: Concept, Pros and Cons, Orchestration, Docker, and Other Alternatives. WWW document. Updated 30 September 2016. Available at: https://medium.com/flow-ci/introduction-to-containers-concept-pros-and-cons-orchestration-docker-and-other-alternatives-9a2f1b61132c [Accessed 18 January 2018].

Mocevicius, R. 2015. CoreOS Essentials. Birmingham: Packt Publishing.

Muikkunen, J. 2018. Business development. Personal talk 26 January 2018. Neos IT Services GmbH.

Negus, C. 2015. Docker Containers: From Start to Enterprise. Boston: Addison-Wesley Professional.

N. Dragoni, S. Giallorenzo, A. Lluch Lafuente, M. Mazzara et al. 2017. Microservices: yesterday, today, and tomorrow. WWW article. Updated June 2016. Available at: https://www.researchgate.net/publication/305881421_Microservices_yesterday_today_and_tomorrow [Accessed 12 December 2017].

Orban, S. 2016. 6 Strategies for Migrating Applications to the Cloud. WWW document. Updated 1 November 2016. Available at: https://medium.com/aws-enterprise-collection/6-strategies-for-migrating-applications-to-the-cloud-eb4e85c412b4 [Accessed 10 January 2018].

Patankar, M. 2015. Moving to Cloud Platform as a Service – Pros and Cons. WWW document. Updated 24 November 2015. Available at: https://www.ibm.com/developerworks/community/blogs/96960515-2ea1-4391-8170-b0515d08e4da/entry/Moving_to_Cloud_Platform_as_a_Service_Pros_and_Cons?lang=en [Accessed 23 January 2018].

Patni, S. 2017. Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS. New York City: Apress.

Peacock, M. 2015. Creating Development Environments with Vagrant - Second Edition. Birmingham: Packt Publishing.

Proffitt, B. 2013. What APIs Are And Why They’re Important. WWW document. Updated 19 September 2013. Available at: https://readwrite.com/2013/09/19/api-defined/ [Accessed 03 February 2018].

Rabeler, C. 2016. Microsoft Azure Essentials Migrating SQL Server Databases to Azure. Redmond: Microsoft Press.

Salehi, S. 2016. Mastering Symfony. Birmingham: Packt Publishing.

Silex. 2018. Usage. WWW document. Available at: https://silex.symfony.com/doc/2.0/usage.html [Accessed 27 January 2018].

Simmons, L. 2016. Api Tutorial for Beginners. WWW document. Updated 15 January 2016. Available at: https://blog.cloudrail.com/api-tutorial-for-beginners/ [Accessed 03 February 2018].

Smiler, K.S., Agrawal, S. 2016. Learning CoreOS. Birmingham: Packt Publishing.

Somasundaram, R. 2013. Git: Version Control for Everyone Beginner's Guide. Birmingham: Packt Publishing.

Stachmann, B., Preißel, R. 2014. Git: Distributed Version Control—Fundamentals and Workflows. Montréal: Brainy Software.

Stoneman, E. 2017. Docker on Windows. Birmingham: Packt Publishing.

Symfony. 2018. The HttpFoundation Component. WWW document. Available at: https://symfony.com/doc/current/components/http_foundation.html [Accessed 3 February 2018].

Thamarai Selvi, S., Vecchiola C., Buyya, R. 2013. Mastering Cloud Computing. Amsterdam: Elsevier Science.

Thompson, C. 2015. Vagrant Virtual Development Environment Cookbook. Birmingham: Packt Publishing.

Thomson, L., Welling, L. 2016. PHP and MySQL Web Development. 5th edition. Indianapolis: Addison-Wesley Professional
Tozzi, C. 2017. Docker Downsides: Container Cons to Consider before Adopting Docker. WWW document. Updated 1 November 2016. Available at: http://www.channelfutures.com/open-source/docker-downsides-container-cons-consider-adopting-docker [Accessed 21 January 2018].

Tozzi, C. 2017. Enterprise Docker. 1st edition. Sebastopol: O'Reilly Media, Inc.

Turnbull, J. 2014. The Docker Book. New York City: Turnbull Press.

Ugia Gonzalez, J.L., Krishnan, S.P. T. 2015. Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects. New York City: Apress.

Vadapalli, S. 2017. Hands-on DevOps. Birmingham: Packt Publishing.

Vagrant. 2018. Introduction to Vagrant. WWW document. Available at: https://www.vagrantup.com/intro/index.html [Accessed 25 January 2018].

van den Berg, M. 2015. Managing Microsoft Hybrid Clouds. Birmingham: Packt Publishing.

van der Woude, W. 2017. Application System Engineer. Email 20 December 2017. Neos IT Services GmbH.

van der Woude, W. 2018. Application System Engineer. Email discussion 5 January – 3 February 2018. Neos IT Services GmbH.

Voss, R., Santacroce, F., Olsson, A., Narębski, J. 2016. Birmingham: Packt Publishing.

Waschke, M. 2015. How Clouds Hold IT Together: Integrating Architecture with Cloud Deployment. New York City: Apress.

Yang, X., Liu, L. 2013. Principles, Methodologies, and Service-Oriented Approaches for Cloud Computing. Hershey: IGI Global.